1. L'objet logiciel

Un objet est une entité informatique, autonome et modulaire, disposant de son propre code de manipulation, les méthodes, et enfermant ses propres données, les attributs. On parle également de fonctions membres et données membres.

Les méthodes traduisent le : "que sait faire l'objet ?", les attributs traduisent le "que connait l'objet ?".

L'analogie avec un objet concret, par exemple un téléviseur, est immédiate un téléviseur dispose de méthodes (boutons de face avant) permettant de l'utiliser, "mise en marche", "changement de canal", "réglage de volume", etc., et comporte en interne les attributs (composants électroniques) nécessaires à son fonctionnement.

On remarquera, et c'est également vrai pour l'objet informatique, que l'utilisation d'un objet suppose de savoir s'en servir mais ne nécessite pas la connaissance intime de son fonctionnement interne.

De plus, dans le cadre d'un développement (informatique ou industriel), la priorité est donnée à la spécification des méthodes, à ce que l'objet "doit pouvoir faire", sans forcément savoir a priori comment le faire. L'implémentation interne sera ensuite la mise en œuvre des spécifications.

2. Classes et instances

Dans la terminologie objets, on appelle classe la famille, le type, la nature de l'objet. On appelle instance une occurrence, une réalisation d'un objet de classe donnée.

"Mon téléviseur" (instance) n'est pas le même objet physique que celui (autre instance) de mon voisin. Et ceci même si les deux appareils sont exactement du même modèle (même classe).

En programmation traditionnelle, ces notions existent sous les appellations type et variable. Ainsi, en Pascal :

```
x, y : real;
n : integer;
```

on déclare deux instances, nommées x et y, de la classe numérique réel et une instance, nommée n, de la classe numérique entier.

3. Mécanismes de spécification

A titre d'exemple (minimaliste) on désire, dans le cadre d'une application géométrique, disposer d'objets Cercle.

L'utilisateur potentiel de l'objet va spécifier les méthodes requises (ces méthodes sont des fonctions informatiques classiques, avec nom, arguments d'appel, type retour) :

- une méthode Move, appelée avec deux arguments réels, permet de déplacer l'objet en relatif.
- une méthode Zoom, appelée avec un argument réel, permet d'appliquer un facteur d'échelle à l'objet.
- une méthode Area, sans arguments, retourne la surface de l'objet.

A partir de ces spécifications, le développeur d'objet va choisir d'implémenter trois attributs numériques, la position X du centre et le rayon du cercle.

NB : dans une bonne conception objet, tout attribut doit pouvoir être justifié relativement à la mise en œuvre d'une ou plusieurs méthodes. C'est pourquoi cette étape s'effectue dans un deuxième temps.

4. Interface de classe

En C++, une interface de classe est une description symbolique de l'objet, destinée au compilateur. A ce titre, elle comporte la définition publique de la classe, méthodes utilisateurs, et l'implémentation privée des attributs (et/ou méthodes internes) nécessaires au fonctionnement.

Traditionnellement, une interface est écrite dans un fichier header séparé, portant le nom de la classe.

Par exemple, ici, cercle. h

```
class Cercle
{
// Méthodes utilisateurs public:
void Move(float deltax, float deltay);
void Zoom(float scale);
float Area();
};
// Attributs implantation private:
float cX, cY;                    // Position centre
float cR;                        // Rayon
```

NB : cet exemple est volontairement réduit à son strict minimum et va être complété au cours de ce chapitre.

5. Instanciations

Le programme application désirant utiliser une classe doit inclure dans son code source le fichier interface, puis déclarer une ou plusieurs instances de cette classe et les manipuler :

```
int main()
{
float S; // Variable locale
Cercle toto; // Instance de cercle
toto.Move(150, 0); /* Deplace toto */
S = toto.Area(); /* Affiche sa surface */
printf("Surface = %f\n", S);
toto.Zoom(1.5); /* Zoom et reaffiche */
S = toto.Area();
printf("Surface apres zoom = %f\n", S);
return 0;
}
```

On remarquera la syntaxe des appels de méthodes. Le nom de la méthode est insuffisant par lui-même, il faut spécifier l'objet auquel on veut l'appliquer, i.e. le nom de l'instance.

On pourrait vouloir manipuler deux cercles :

```
Cercle c1, c2;
c1.Zoom(3); c2.Zoom(0.5);
float S1, S2;
S1 = c1.Area();
S2 = c2.Area();
```

Auquel cas la surface de c1 n'a aucune raison d'être la même que la surface de c2. C'est le travail du compilateur d'assurer que les méthodes seront appelées sur les objets appropriés.

NB : contrairement à C, C++ autorise les déclarations, scalaires ou objets, à n'importe quel endroit du code et non plus uniquement en en-tête de fonction. Cf. ci-dessus la déclaration de S1 et S2.

6. Implémentation de classe

L'exemple ci-dessus montre que l'on est capable d'écrire du code source utilisant des objets sans se préoccuper de l'implantation. (On peut savoir utiliser un téléviseur sans être électronicien, sans savoir le construire !) L'interface et la connaissance de ce que font les méthodes est suffisante.

Pratiquement, il faudra tout de même assurer l'implémentation de l'objet et l'écriture effective du code des méthodes. Traditionnellement, un objet est encodé dans un fichier source portant le nom de la classe.

Par exemple, ici, cercle.cpp

```
/* Deplacement */
void Cercle::Move(float deltax, float deltay)
{
cX += deltax; cY += deltay;
}
/* Echelle */
void Cercle::Zoom(float scale)
{
cR *= scale;
}
/* Surface */
float Cercle::Area()
{
return 3.14159 * cR * cR;
}
```

L'exemple est suffisamment simpliste pour que la programmation n'appelle pas de commentaire particulier. On notera simplement la syntaxe de définition d'une méthode :

```
NomClasse::NomMethode
```

Dans le code des méthodes, les attributs sont invoqués sous leur nom symbolique, tel que défini dans la déclaration d'interface. En pratique, chaque instance d'une classe dispose de son propre jeu d'attributs et c'est au compilateur C++ de faire en sorte que les données convenables soient manipulées.

Ainsi, lorsqu'un code utilisateur manipule plusieurs instances :

```
Cercle c1, c2;
c1. Zoom (3. 0) ; c2.Zoom(0.5);
```

c'est bien l'attribut cR de l'instance c1 qui sera multiplié par 3, et l'attribut cR de l'instance c2 qui sera divisé par 2.

Le code d'implémentation d'un objet est donc une programmation générique, se référant à "l'objet courant pour l'appel".

7. Visibilités

On aura remarqué, dans la description d'une interface de classe la présente de deux attributs de visibilité, **public** et **private**.

Pour des raisons liées à l'implémentation des compilateurs C++, une interface de classe doit comporter la description exhaustive de l'objet, méthodes publiques mais aussi tous les attributs et méthodes privées. Les attributs de visibilité ont pour rôle de signaler au compilateur les invocations autorisées ou interdites :

- le code interne à l'objet, i.e. la programmation des méthodes de classe, a accès à tout, public ou privé !
- le code externe, en général le code utilisateur, n'a accès qu'aux invocations publiques. L'accès explicite à un attribut privé n'est pas autorisé :

```
Cercle c1;
c1.Zoom(2.0);                    // Ok, méthode publique
c1.cR *= 2.0;                    // Erreur en compilation !
```

Ce mécanisme, appelé aussi encapsulation des données, est là pour obliger le code utilisateur à passer par les méthodes prévues par le concepteur. Ce n'est pas une contrainte mais plutôt une sécurité.

D'une part, le concepteur de l'objet qui assure la maintenance des fichiers relatifs à l'objet, fichier interface machin.h et fichier source machin.cpp, reste libre de modifier l'implémentation interne, de changer des noms d'attributs, d'en ajouter, d'en supprimer. Le code utilisateur reste valide (à une recompilation près) tant que l'interface publique est stable (et, en principe, elle doit l'être puisqu'elle résulte d'une spécification initiale).

D'autre part, et c'est souvent le cas en pratique, un objet important exige une certaine cohérence sur ses attributs. Parfois, une simple modification de valeur peut nécessiter un recalcule d'autres attributs. En obligeant le code utilisateur à passer par des "guichets", le concepteur est assuré de pouvoir maintenir un objet consistant.

Par défaut, dans une classe C++, tout est privé. Ces attributs sont à placer aux endroits ad-hoc de la définition de classe, on peut les alterner, la visibilité est valide pour "tout ce qui suit" jusqu'au prochain attribut.

NB : Il existe d'autres mécanismes de protection.

8. Cycle de vie

Un objet logiciel a une vie et une mort. Le langage offre la possibilité d'implémenter deux méthodes de classe particulières, le constructeur et le destructeur. La compilation assure que le constructeur, s'il est spécifié, sera appelé lors de la création d'une instance et avant la toute première utilisation, et que le destructeur, s'il est spécifié, sera appelé juste avant la destruction physique de l'objet.

Ainsi, dans l'exemple ci-dessus, on ne sait pas, lorsqu'on instancie un objet cercle, ce que valent ses attributs. Le rôle typique d'un constructeur est d'assurer une initialisation par défaut qui soit convenable.

Ces méthodes se déclarent sans type et avec un nom imposé, NomClasse pour un constructeur, ~NomClasse pour un destructeur.

L'interface de classe (fichier cercle. h) deviendrait :

```
class Cercle
{
// Constructeur, destructeur public:
Cercle();
~Cercle();
// Methodes utilisateurs
public:
void Move(float deltax, float deltay);
etc ...
```

et l'implémentation (fichier cercle. cpp) comporterait :

```
/* Constructeur */ Cercle::Cercle()
{
cX = cY = 0.0;                    // A l'origine par défaut
cR = 1.0;                         // Rayon unité
1
/* Destructeur */
Cercle::~Cercle()
{
// Rien à faire de spécial !
}
```

Le mécanisme d'initialisation par constructeur est, en pratique, indispensable, c'est le moyen d'assurer que tout objet commence sa vie avec une configuration plausible. Dans un environnement objets bien conçu, les mécanismes d'instanciation doivent toujours produire des objets utilisables en l'état et le code utilisateur ne devrait avoir à reconfigurer que lorsque les valeurs par défaut sont inacceptables.

Un autre mécanisme très intéressant est la possibilité qu'a le concepteur d'objet de proposer différents constructeurs de classe. Par exemple, on pourrait proposer un constructeur de cercles prenant une valeur initiale de rayon en argument.

Interface de classe :

```
class Cercle
{
     // Constructeurs, destructeurpublic:
     Cercle();
     Cercle(float rayon);
      Cercle();

     ...
 }
```

Implémentation :

```
/* Constructeur par defaut */ Cercle::Cercle()
{
cX = cY = 0.0;                    // A l'origine par defaut
cR = 1.0;                         // Rayon unite
1
/* Constructeur avec rayon */ Cercle::Cercle(float rayon)
{
cX = cY = 0.0;                    // A l'origine par defaut
cR = rayon;                       // Rayon utilisateur
}
```

Le code utilisateur pourra ensuite instancier des cercles en spécifiant tel ou tel constructeur :

```
Cercle c1; // Construction par defaut, rayon 1.0
Cercle c2(5.5)// Construction avec argument rayon
```

L'appel correct de tel ou tel constructeur sera effectué par le compilateur C++, en fonction du contexte de déclaration et ce, de manière transparente pour le code utilisateur. On n'appelle jamais explicitement un constructeur ou un destructeur :

```
Cercle c1;                        // Le constructeur est appelé ici
c1.Cercle();                      // Non !
```

L'implantation d'un destructeur (qui lui, ne peut être multiple ni comporter d'arguments) est moins fréquente. Dans l'exemple ci-dessus elle est inutile puisqu'on ne fait rien. Elle se justifie lorsqu'un objet nécessite des tâches de "ménage" lors de sa destruction : fermeture d'un fichier qui aurait été ouvert par l'objet, libération d'une zone mémoire allouée, etc. Comme le destructeur est également appelé automatiquement, on est ainsi sûr de ne rien oublier.

9. Gestion des objets

Le langage C dispose de trois mécanismes de durée de vie des données :

- statique ou global : la durée de vie des données est celle du programme.
- automatique ou local : la durée de vie est limitée à un contexte { ... } de programme.
- dynamique : les données sont crées (malloc ()) et détruites explicitement (free()), leur manipulation s'effectue via un pointeur.

Les mêmes mécanismes sont utilisables en C++, pour les variables classiques et, a fortiori, pour des objets.

9.1. Gestion directe

```
Cercle c1;          // Global, statique

void toto ( )

{
     Cercle c2;     // Local
     ...
     if( ... ) {    // Nouveau contexte
        Cercle c3; // Local
         ...
         ...
     }              // Fin de contexte, c3 est detruit
     ...
 }                  // Fin de fonction, c2 est detruit
```

Un objet est créé (i.e. la mémoire est allouée puis le constructeur est appelé) lors de sa déclaration. Un objet global (c1 dans l'exemple ci-dessus) est construit au lancement du programme.

Un objet est détruit (i.e. le destructeur est appelé puis la mémoire est libérée) lorsque qu'il devient hors de portée, lorsque l'on quitte le contexte qui le déclarait. Un objet global est détruit en fin de programme.

NB : on voit ici toute la puissance apportée par la possibilité de programmer un destructeur lorsqu'un objet nécessite des tâches de "ménage", libération de ressources et autres : un simple return, n'importe où dans une fonction, fera quitter le con- texte et invoquera implicitement toutes les opérations de nettoyage sur tous les objets locaux existants.

9.2. Gestion indirecte

La gestion de variables par pointeurs, familière aux programmeurs C, s'effectue de manière identique en C++ :

- déclaration d'un pointeur typé
- allocation d'une donnée
- manipulation classique via l'opérateur -> ou l'opérateur d'indirection *, cf. exemple ci-dessous
- enfin, destruction explicite

```
float S;
Cercle *pc;                              // Pointeur de Cercle
pc = new Cercle; // Allocation
pc->Zoom(3.5);                           // Appel methode
(*pc).Zoom(5.0); // Autre syntaxe possible
S = pc->Area();                          // Appel methode
...
delete pc;                               // Liberation
```

La différence porte sur la mise en œuvre des allocations et libérations. En C++ on n'utilise JAMAIS les appels malloc () et free() !

L'opérateur new effectue l'allocation mémoire pour l'objet ET l'appel du constructeur. L'opérateur delete effectue l'appel du destructeur puis la libération mémoire. Dans le cas de constructeurs multiples, c'est la syntaxe de création qui lève les ambiguïtés :

```
Cercle*pc1,      *pc2;
*pc1 =new Cercle;              //  Constructeur par defaut
*pc2 =new Cercle(5.0);         //  Constructeur "avec rayon"
```

A noter que ces opérateurs sont également utilisables sur des variables simples et existent sous une forme "tableaux" :

```
int *pi; float *pf;
pi = new int;                                  // Allocation scalaire
pf = new float[50]; // Allocation tableau ...
delete pi;                                     // Liberation scalaire
delete[] pf;                                   // Liberation tableau
```

Remarquer la notation delete [ ] pour détruire un tableau. On peut manipuler des tableaux de scalaires mais aussi des tableaux d'objets. Dans le cas de tableaux d'objets, les appels des constructeurs et destructeurs sont effectués sur tous les éléments du tableau.

NB : tout comme en C, un pointeur déclaré dans une fonction est détruit (en tant que variable pointeur) lorsqu'on quitte le contexte, mais la mémoire éventuellement associée n'est pas libérée !

En C++, en cas de création d'objets par **new**, on devra assurer la destruction explicite par **delete**.